LLNL-JRNL-422668

Massively Parallel Acceleration of a Document-Similarity Classifier to Detect Web Attacks

Craig Ulmer[b], Maya Gokhale[a,*], Brian Gallagher[a], Philip Top[a], Tina Eliassi-Rad[a]

[a] *Lawrence Livermore National Laboratory*
[b] *Sandia National Laboratories, CA*

Abstract

This paper describes our approach to adapting a text document similarity classifier based on the Term Frequency Inverse Document Frequency (TFIDF) metric to two massively multi-core hardware platforms. The TFIDF classifier is used to detect web attacks in HTTP data. In our parallel hardware approaches, we design streaming, real time classifiers by simplifying the sequential algorithm and manipulating the classifier's model to allow decision information to be represented compactly. Parallel implementations on the Tilera 64-core System on Chip and the Xilinx Virtex 5-LX FPGA are presented. For the Tilera, we employ a reduced state machine to recognize dictionary terms without requiring explicit tokenization, and achieve throughput of 37MB/s at slightly reduced accuracy. For the FPGA, we have developed a set of software tools to help automate the process of converting training data to synthesizable hardware and to provide a means of trading off between accuracy and resource utilization. The Xilinx Virtex 5-LX implementation requires 0.2% of the memory used by the original algorithm. At 166MB/s (80X the software) the hardware implementation is able to achieve Gigabit network throughput at the same accuracy as the original algorithm.

Keywords: cybersecurity, document classification, machine learning, multi-core, reconfigurable computing,

[☆]This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

[*] *Contact:* 925-422-9864 (ph), 925-423-2993 (fax) (Maya Gokhale)
Email address: maya@llnl.gov (Maya Gokhale)

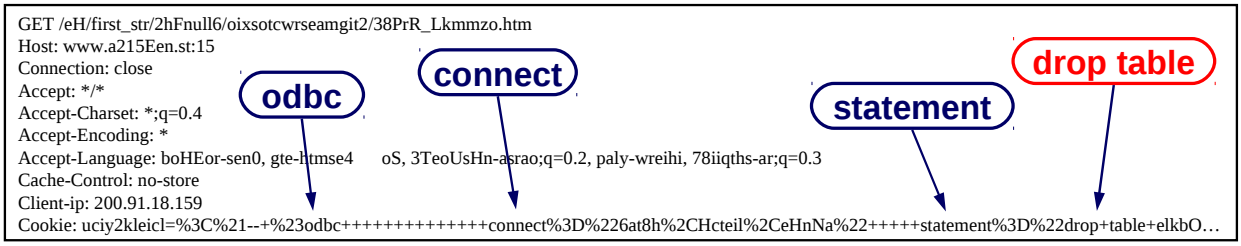

Figure 1: Sample HTTP request with malicious database activity

1. Introduction

As the world becomes more reliant on Web applications for commercial, financial, and medical transactions, cyber attacks on the World Wide Web are increasing in frequency and severity. Web applications provide an attractive alternative to traditional desktop applications due to their accessibility and ease of deployment. However, the accessibility of Web applications also makes them extremely vulnerable to attack. This inherent vulnerability is intensified by the distributed nature of Web applications and the complexity of configuring application servers. These factors have led to a proliferation of Web-based attacks, in which attackers surreptitiously inject code into HTTP requests, allowing them to execute arbitrary commands on remote systems and perform malicious activities such as reading, altering, or destroying sensitive data (e.g., credit card numbers, trade secrets, medical history). Figure 1 shows a sample HTTP request containing a "SQL Injection" attack, in which an entire data table is deleted by a malicious SQL command.

In order to prevent such attacks, we need to identify malicious code in incoming HTTP requests and eliminate bad requests before they are processed. Using machine learning techniques, we can build a classifier to automatically label requests as "Valid" or "Attack." For this study, we develop a simple, but effective HTTP attack classifier, based on the vector space model [1] used commonly for Information Retrieval. Our classifier not only separates attacks from valid requests, but can also identify specific attack types (e.g., "SQL Injection" or "Path Traversal").

The classification performance of our sequential algorithm [2] compares favorably to previously published work on the ECML/PKDD 2007 Discovery Challenge data set, achieving an F1 score of 0.93, compared to 0.80 and 0.48 for previous approaches. Our approach demonstrates accuracy of 94%.

Building on this approach, we have devised two massively parallel, *streaming* algorithms to classify HTTP requests as they arrive in a text stream transmitted over a high-speed internetwork. Using our methods, real time classification of HTTP requests into attack vs. normal categories serves as an advanced intrusion prevention method capable of detecting and thwarting web attacks as they occur.

We present a parallel implementation optimized for the Tilera^{TM} 64-core System on Chip. The two level data parallel approach organizes the cores into multiple units of nine tiles (cores). This nine-tile processing unit is then replicated to the capacity of the chip, allowing the units to process independent document streams. By compacting the term dictionary to fit into the L2 cache of each core, we can classify web request documents at 37MB/s (18.5X speedup over the sequential algorithm).

Our FPGA algorithm has been optimized for streaming computation in a hardware pipeline that exploits on-chip distributed RAM (known as Block RAM or BRAM on Xilinx FPGAs) for high performance. The hardware building blocks are highly configurable, and their configuration parameters are generated semi-automatically through analysis of the training data set. The classifier demonstrates throughput of 166MB/s (80X speedup over the sequential algorithm) at the same accuracy as the sequential algorithm.

2. Sequential Algorithm

A novel HTTP attack classification algorithm was proposed by Gallagher and Eliassi-Rad [2], and a sequential Java implementation was devised. This algorithm is based on a classic vector space model from Information Retrieval. Vector space models (a.k.a. term vector models) were first used in an information retrieval system called SMART [3]. These models are commonly used to retrieve information relevant to an input query from a text/document corpus. The first Web search engines (i.e. before Google) were predominately based on vector space models. Over the years, several vector space classification methods have been invented, two of the most popular being Rocchio and kNN classification (see Sections 14.2 and 14.3 of [4]).

2.1. Document Similarity

The HTTP attack classification algorithm depends on the concept of document similarity. Document similarity using term weights is a well-understood Information Retrieval technique [1]. The goal is to: (1) weight

terms in each document such that the most representative terms receive the highest weight, (2) represent each document by a vector of term weights, and (3) compare documents to one another using a similarity measure over the space of term vectors.

An effective and well-known weighting scheme for this purpose is $tfidf$ [1]. The $tfidf$ weight consists of two components: tf and idf. Term frequency (tf) measures how often a specific term t occurs within a given document d, relative to all terms v in d:

$$tf(t,d) = \frac{count(t,d)}{\sum_{v \in d} count(v,d)} \tag{1}$$

Inverse document frequency (idf) measures the proportion of documents d in the collection D in which term t occurs.

$$idf(t) = \frac{log\,|D|}{|\{d \in D : t \in d\}|} \tag{2}$$

The $tfidf$ assigns the highest weight to terms t that occur frequently in document d, but occur in few other documents.

$$tfidf(t,d) = tf(t,d) \cdot idf(t) \tag{3}$$

In the vector space model, document d is characterized by $tfidf$ vector V_d. Each component i of V_d holds the $tfidf$ score of the i'th term in the document collection. Similarity between documents d and a is calculated using the cosine of the angle θ between the $tfidf$ vectors V_d and V_a:

$$sim(d,a) = cos(\theta_{V_d,V_a}) \tag{4}$$

i.e.

$$sim(d,a) = \frac{V_d \cdot V_a}{\|V_d\|\|V_a\|} \tag{5}$$

or equivalently

$$sim(d,a) = \frac{\sum_{t \in d \cap a} tfidf(t,d) \cdot tfidf(t,a)}{\sqrt{\sum_{t \in d} tfidf(t,d)^2}\sqrt{\sum_{t \in a} tfidf(t,a)^2}} \tag{6}$$

2.2. Data Set

Our experiments primarily use a data set released by the European Conference on Machine Learning (ECML) and the 11th European Conference on the Principles and Practice of Knowledge Discovery in Databases (PKDD) in 2007 as part of the 2007 Discovery Challenge [5]. The data set contains more than 120,000 labeled HTTP requests, of which 50,000 are for training, and 70,000 constitute the testing data. 70% of the training requests are normal, valid requests, and 30% contain cybersecurity attacks. In the testing data, 60% are normal and 40% are attack.

There are seven different sorts of attacks: cross site scripting (XSS), SQL injection, LDAP injection, XPATH injection, directory path traversal, command execution, and Server Side Include (SSI) attacks. The 2007 challenge was to detect and characterize the attacks.

Each training and test instance in the data set contained the full text of the http request, divided into the following components: method, protocol, uri, query, headers, and body. In addition, each HTTP request included the following contextual attributes:

- Operating system running on the Web Server (UNIX, WINDOWS, UNKNOWN)
- HTTP Server targeted by the request (APACHE, MIIS, UNKNOWN)
- Is the XPATH technology understood by the server? (TRUE, FALSE, UNKNOWN)
- LDAP database on the Web Server? (TRUE, FALSE, UNKNOWN)
- SQL database on the Web Server? (TRUE, FALSE, UNKNOWN)

In addition to the ECML/PKDD data set, we have applied our algorithm to the 2009 Inter-Service Academy Cyber Defense Exercise data, the "West Point" data set [6]. This data was generated by an experiment to have the National Security Agency attack a network at West Point Academy. This data set was used to analyze throughput performance of our algorithms. Since it was not labeled, we could not derive accuracy measures from the West Point data set.

2.3. Related Work

Discovery Challenge Over 25 groups registered for the ECML/PKDD Discovery Challenge, but only two submitted final results. According to Rassi et. al. [7], most researchers failed to submit results because they found that traditional data mining approaches were unsuccessful and felt that a specialized knowledge of attack detection was required to adequately address the problem. The two groups that submitted results took very different approaches to the problem. Pachopoulos et al. [8] tried two different approaches:

Approach 1: Extract binary features from the HTTP request data, perform feature selection, and then apply C4, a standard supervised-learning decision-tree algorithm, to build a classifier.

Approach 2: Use the string representations of the various HTTP fields directly as features for input to a Support Vector Machine using a String Kernel. The authors abandoned the SVM approach in favor of the C4 approach after the former failed to deliver satisfactory performance.[1] The binary features used as input to C4 are based on the presence or absence of a number of attack indicators, derived manually by the authors.

The approach of Exbrayant [10] is based on constructing a language model that is used to define HTTP attack patterns. The author notes that attack patterns consist of sequences of keywords, variables, and symbols. Thus, the approach derives rules based on such sequences that can be used to identify the beginning and end of attack strings in HTTP requests. The core of this approach involves extracting and evaluating potential rules. Once this is done, it is straightforward to classify a candidate request based on whether it matches a given rule. While the Exbrayant classifier shows better results than Pachopoulos, it does not achieve the accuracy of our TFIDF approach.

TFIDF in hardware Chen et. al. [11] report on an FPGA design combining on-chip general purpose processor with an array of term counting IP blocks. In contrast to our design which uses Bloom filters, the implementation uses fixed length comparators to compare chunks of the term in successive cycles. Simulation results are presented that show a 3–7 times speedup over software. The work performs term frequency counting only.

Network Intrusion Detection There is a large body of literature reporting hardware and multi-core acceleration of network intrusion detection. The most common approach is to compile signatures expressed as regular expres-

[1] Pachopoulos et al. [8] used WEKA's implementation of C4 and SVM [9].

sions into FPGA hardware, eg. TCAM [12] or logic [13]. Network intrusion detection using Principal Components Analysis is reported by Das et. al. [14]. In contrast, our work classifies based on term frequency rather than fixed regular expression patterns.

2.4. TFIDF Method

Our approach applies the TFIDF method to train and run a classifier to detect attacks. To train the classifier, all requests of a specific attack type in the training set are combined into a single document, resulting in eight reference attack documents. $tfidf$ vectors are computed for each reference document.

In the testing phase, each HTTP request in the testing data set is considered to be a single document, and the cosine similarity between the incoming request and each reference attack document is computed. Once all similarity scores are computed, a threshold operation is applied to remove terms whose similarity scores are lower than a threshold. This operation improves the quality of the results when comparing the attack scores to the valid score, most notably when the dictionary has a large number of terms. This parameter is typically set during training to balance precision and recall [15] statistics. A lower value implies a more hostile environment, where it is important to filter anything that appears malicious, even if it means mistakenly filtering harmless traffic. The document is classified as belonging to the class of the most similar reference document. The process is shown in Figure 2.

On the ECML/PKDD 2007 Discovery Challenge data set, our sequential algorithm gave accuracy of 94% in distinguishing attack vs. normal and 91% in correctly distinguishing the type of attack. The single-threaded Java software implementation of the algorithm runs at about 2MB/s on a standard PC workstation (2.2GHz, quad core, 8GB memory). A Hadoop MapReduce implementation of this algorithm, including the original Java source code, is publicly available [16].

We note that although the TFIDF approach compares favorably to other state-of-the-art approaches for Web attack classification, it does have some limitations, which we address here. The single data characteristic that most impacts the approach is how reliably specific terms map to concepts of interest. It may be possible to evade detection by carrying out an attack using an unusual set of terms. However, the regularity of programming language syntax works in our favor here. For instance, there are a small number of SQL commands that result in the deletion of a database table. Of course, if

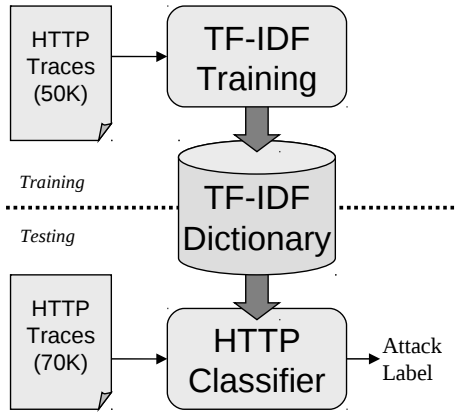

Figure 2: Training and testing the classifier

commands are obfuscated (e.g., via convoluted JavaScript), this will create problems for any of the proposed detection techniques. Another limitation of the TFIDF-based approach is that the discovery of new attacks potentially requires expanding the vocabulary and relearning attack models. Limiting assumptions of the approach include term independence and irrelevance of term order. Ideally we would model meaning at a higher-level than term frequency. However, as we saw with the language modeling approach in section 2.3, a more sophisticated model does not necessarily lead to better results in practice.

3. Parallel Implementations

The TFIDF method offers a novel approach to detecting malicious web HTTP requests. However, the sequential algorithm implementing Equation 6 requires access to the entire document collection a priori. Further, at 2MB/s the sequential algorithm cannot process a real-time data stream. Since our goal is a streaming, real-time approach, we study the equation and data set for optimization opportunities. We seek to optimize the classification phase of the TFIDF algorithm, and perform the training phase off-line.

Streaming In a streaming environment, the data must be processed in a single pass with very limited buffering. We observe that according to Equation 6 a term's *idf* scores can't be computed until the entire document collection has been scanned. In a streaming environment, the document collection is

never complete, and therefore the *idf* we used is based on the training document collection, which is pre-computed and can simply be looked up in the dictionary. Both parallel implementations use an *idf* value pre-computed from the training data set.

Memory reduction To maintain high throughput, the classifier model (called dictionary in this context) must be stored on-chip. In this application, the limited memory presents a formidable challenge, as the full dictionary for the ECLM training data set uses 47MB. While this is fairly modest in a workstation environment, it is not feasible when on-chip memory is distributed among many parallel processing cores.

The classifier model or dictionary holds the $tfidf$ score for each term encountered in the training data set. The number of terms in the dictionary ultimately dictates how much information is available during classification. A reasonable accuracy can be achieved with only a small number (e.g., 32) of high-value keywords. Supplementing these terms with a large number (e.g., hundreds to thousands) of less important but still relevant terms typically improves accuracy until the classifier becomes overtrained. After this point accuracy may stay constant or degrade. Increasing the number of dictionary terms increases the amount of data a classifier must maintain.

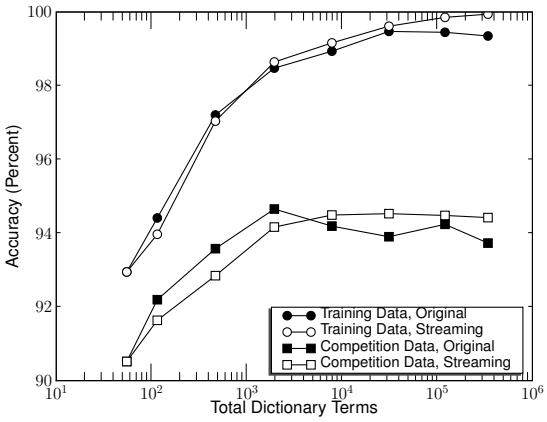

Figure 3: Impact of number of terms on accuracy

A test program was constructed to evaluate the impact of changes to the original algorithm and observe the impact of reducing the number of terms in the dictionary A dictionary was constructed based on the training data set.

We varied the number of terms in the dictionary and then measured accuracy when the original and streaming versions of the algorithm were used to classify the training and testing data sets. The results are presented in Figure 3. As expected, the classifier performed exceptionally well on the training data set, with accuracy improving as more terms are included. The testing results level off after two to eight thousand unique terms. This plateau is expected and indicates the point at which the classifier becomes overtrained. Both parallel implementations truncate the number of terms to reduce the size of the dictionary.

4. Tilera Mapping

4.1. Tilera Architecture

The Tilera processor is a low power, many-core System on Chip. The Tilera 64 used in this application consists of an 8×8 array of 700MHz custom 32-bit integer processors running Linux. The processors communicate with each other, memory, and external devices through a unique two-dimensional switched mesh interconnect with five communication networks. The chip includes a 10Gb/s Ethernet port, PCI Express ports, and a DDR2 memory controller. Like conventional multi-core processors, the Tilera cores (or "tiles") have a shared address space with hardware maintained cache coherence. Unlike conventional multi-core processors, the tiles include switches to route communication. The processors communicate over a dynamic on-chip interconnection network or a user-managed static interconnection network. The switches also transparently route cache, memory and I/O accesses. Each tile has two 8KB L1 caches for program and data, and a 64KB L2 cache for local scratchpad operations. The tile has a 3-way VLIW instruction set for concurrent memory, I/O, and ALU operations, DSP-like instructions, a 2-stage instruction pipeline, and sixty-four 32-bit registers.

The tiles are programmed in C or assembly language. The Tilera runs Linux and supports pthreads and sockets as well as a proprietary high performance concurrency and communication library.

4.2. Tilera Algorithm

The Tilera algorithm is shaped by the limited cache available to each tile. Although hardware memory management gives a unified address space

across the tiles, a variable latency of memory accesses can dramatically reduce streaming throughput. Thus, the principal design decision is to reduce the model size to fit the dictionary into the 64KB L2 cache of each tile.

Using accuracy results described above, we truncate the term frequency vector to 255 elements, for an accuracy of 92%. This allows a complete dictionary to be stored on each tile, and eliminates inter-tile communication to access dictionary elements. The eight dictionaries (seven attack types and "normal") are stored in eight tiles, with a ninth tile dedicated to data ingest. The unit of nine tiles is replicated six times, using 54 of the 64 tiles plus an aggregator tile. Further replication is not possible as several tiles are reserved for the OS. The algorithm's spatial layout is shown in Figure 4. The architecture demonstrates two levels of data parallel processing. First, eight different attack classifiers analyze the same data in parallel. Second, six different processing units classify different data streams.

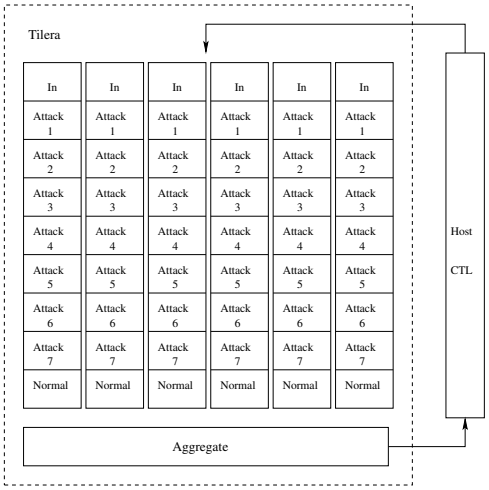

Figure 4: Layout of processes on Tilera cores

A second optimization is to eliminate the overhead of tokenization. Rather than separate the input stream into tokens and then look up each token (term) via hashing into a table, a state transition matrix is used. This matrix is the Tilera algorithm's representation of the dictionary. Each incoming character in the data stream indexes into the state machine transition matrix, yielding the next state. Most state transitions require a table lookup and a conditional. If a delimiter token occurs, the count for the appropriate term is incremented. When an end-of-document marker is detected, the similarity

State Machine Structure

	0	1	2	3	4	5	6	7	8	9	10
A	0	0	4	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	9	0	0	0	0
C	0	0	0	0	0	0	0	0	6	1*3	0
D	0	2	0	6	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	3	0	8	5
F	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	5	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	10	0	0
N	0	0	0	0	0	9	0	0	0	0	0
O	0	3	0	0	10	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	1*2
Q	0	0	0	0	0	0	0	0	0	0	0
R	0	0	4	0	0	0	0	0	0	0	0
S	0	7	0	0	0	0	0	0	0	0	0
T	0	0	0	0	9	0	1*1	2	0	1*4	0
U	0	0	0	0	0	0	0	0	0	0	0
V	0	0	0	0	0	0	0	0	0	0	0
W	0	0	0	0	0	0	0	0	0	0	0
X	0	0	0	0	0	0	0	0	0	0	0
Y	0	0	0	0	0	0	0	0	0	0	0
Z	0	0	0	0	0	0	0	0	0	0	0
space	1	1	1	1	1	1	1	1	1	1	1

Figure 5: Dictionary traversal using state machine

measure is computed and returned to the host.

Figure 5 shows an example of a state transition matrix for the dictionary {Select, Drop, Odbc, Statement}. Each row represents an input character, and each column represents a state. If the word "drop" were given, the state sequence would be (1, 2, 4, 10). If a delimiter were encountered after the "p," the token would be recognized, the term frequency count for the word "drop" would be incremented, and the state machine would return to state 1.

The state machine has a small probability of a false positive in the matching of tokens. Since the number of matches is expected to be small, even a small number of false positives could alter the detection probabilities. To reduce the possibility of false positives, the length of each term is stored on each tile, and a further check performed to make sure the length matches when a matching token is encountered. With this further step the probability of a false positive is greatly reduced. It is not eliminated entirely but no examples of false positives have been observed in the data set.

In our experimental system, the Tilera card is installed in a quad socket, dual core workstation, for a total of eight cores. The host processor and Tilera communicate over a PCI Express bus. The multi-threaded host program send data blocks to the six ingest tiles on the Tilera. The data is sent using the optimized zero copy mechanism in the Tilera. The data ingest tiles then

broadcast the data stream to the processing tiles using the iLib messaging interface in the Tilera. To maintain coherence between data packets sent to the Tilera and the results returned, the messaging blocks are annotated with id codes generated by the host, allowing the host CPU to match a document with the resulting TFIDF similarity score. The host compares scores from the eight classifiers in each processing unit, and uses threshold parameters to determine the document's final classification.

State machine representation The state machine is implemented as a 2D array of 16 bit integers. The lower 8 bits are used as the next state transition index. The upper 8 bits are used as the token id. The reserved token id of 0 is used as the "no match" token. Thus the 16 bit array allows 255 different tokens to be matched.

Each input character in the data stream is truncated to 7 bits for ASCII text packets (though the approach could be expanded to 8 bits if necessary). There are 228 possible states with 128 potential different state transitions to be stored on each tile. The state machine has 2 special states. State 0 is a waiting state that the system enters when there is no possibility of a match and the system is waiting for a new token to start. State 1 is the starting state. Upon receipt of a delimiter token such as space or tab, the system increments the term's count if a term was matched, and then goes to the starting state.

Generating the state machines The state machines are generated off-line, using the top 255 TFIDF scores of each attack type. The algorithm indexes into an array for each character in a term to detect whether a state transition has been defined for the character. If existing paths are in place, the paths are followed until the point at which the state transitions are unspecified. The transitions for the term being encoded are then randomly assigned. Additional checks are performed in the encoding to ensure the term's state transition sequence does not fold on itself or interfere with other terms. At the end of the character sequence, a flag is set to trigger an output with the appropriate term id if the following character is a delimiter (space, tab, ...). The state machine is set up to allow any number of different delimiters to be used with no decrease in performance. With simple modifications to the appropriate generator, the state machine could allow wildcard characters, character classes (eg. numbers, lowercase letters only) and combinations of these patterns.

Beyond the first two states, all states are randomly assigned during the array creation process. For example, suppose the word "Select" must be

Table 1: Throughput results for 1 – 8 attack types with 1–25 processing units

Num. Attacks / Num Units	**1**	**2**	**4**	**8**
1	6.42	6.42	6.42	6.42
2	12.79	12.79	12.77	12.70
3	19.09	19.09	19.09	18.99
4	25.34	25.34	25.34	25.22
5	31.58	31.58	31.58	31.39
6	37.70	37.70	37.58	**37.4**
8	49.84	49.84	49.84	
10	61.87	61.87	61.87	
12	73.55	73.55		
15	90.67	90.67		
20	118.17			
25	**136.57**			

placed in the array. Starting from state 1, we look up the location corresponding to upper or lower case "s." If it is occupied we follow it to the next state already assigned from a previous word. If it is not occupied, we randomly assign the next state to the "s" location and move to that state. The same process is followed in the new state for the "e" and so on. The next state assignment is placed in the lower byte of the array element, the upper byte being used to flag token match with the appropriate term id (from 1 to 255). Once a term is completely assigned, the array is checked for conflicting matches, or patterns that loop on themselves. If either of these conditions occur, the process is re-started with new random state assignments. Finally the entire set is checked for correctness and to ensure no match conflicts. The random assignment ensures an even distribution of the patterns across the entire array, and reduces the probability of false positives.

4.3. Tilera Results

The Tilera algorithm has been devised to allow maximum flexibility to trade off between the number of attack types and throughput. Table 1 shows throughput results for various numbers of attack types ranging from 1 to 8 (columns) versus the number of processing units (1 to 25). The maximum

throughput to detect one attack type using 25 tiles is 136.57MB/s. The standard configuration of eight attack types using six processing units delivers throughput of 37.4 MB/s. For dictionaries with more than eight attack types, the size of each processing unit would increase, and the number of processing units would commensurately decrease.

5. FPGA Hardware Algorithm

The hardware algorithm seeks to minimize the number of multiply and divide operators and to minimize memory usage.

Minimize calculation In our application we generate the eight attack type scores for an input document and label the document based on the largest score. There is an opportunity to simplify Equation 6 and reduce the number of arithmetic operations required to compute the similarity score. There are two places where normalization is applied to allow the scores of one document to be compared to another's scores:

- the tf component of the numerator's $tfidf$ contains a normalization by the number of terms in the input document (expressed as $\sum_{v \in d} count(v, d)$ in Equation 1).
- the denominator's $\sqrt{\sum_{t \in d} tfidf(t, d)^2}$ normalizes the $tfidf$ score across the document collection.

The normalization factors enable comparison between documents. Since these factors are constant across all attack types for a single document, they can be omitted without affecting the classification.

With these simplifications it is possible to translate Equation 6 into a form that can be facilitated through table lookups.

$$sim(d, a) = \frac{\sum_{t \in d \cap a} count(t, d) \cdot C_1 \cdot C_2}{C_3} \tag{7}$$

where

$$\begin{aligned} C_1 &= idf(t) \\ C_2 &= tfidf(t, a) \\ C_3 &= \sqrt{\sum_{t \in a} tfidf(t, a)^2} \end{aligned} \tag{8}$$

Given that all three constants are relative to a particular attack type, information can be combined to reduce table lookup size. In our approach

we combine C_2 and C_3 and refer to the value as a $categoryWeight$. Thus the classifier model requires nine statistics for every term: a single C_1 value to indicate the term's idf and eight $categoryWeight$ values to indicate how relevant the term is to a particular attack type.

Minimize memory As stated in Section 3 the full dictionary for the ECML training data set uses 47MB. We seek to encode the dictionary in a combination of logic and memory of approximately 128KB, a compression factor of more than 367. Three different optimizations are employed to help reduce the dictionary memory footprint to 0.2% the size of the original. First, as discussed above, we truncate the dictionary so that only the N most significant terms per attack type are utilized. N is a parameter to the hardware algorithm, and in the experiments discussed in Section 5.5 is around 1900 terms. Adaptations to the hardware algorithm resulted in comparable accuracy to the sequential one. Differences between the original and streaming runs can be partially attributed to the fine tuning of the threshold operation (see Section 5.3 for a description of thresholding).

The second optimization is to quantize the $categoryWeight$ and idf data values in the dictionary in order to simplify the numerical diversity and allow better information compression. Finally, we utilize a hashing technique that employs an array of Bloom filters to represent the dictionary data. Each of these two optimizations is discussed in detail.

5.1. Quantize Term Scores

TFIDF training generates a large amount of statistical data that is encoded in a dictionary and utilized at runtime to determine the relevance of a document's terms to particular attack. This training typically exposes a small number of keywords that are assigned a high $categoryWeight$ while the majority of data values receive much lower values. The log histogram for one attack document's $categoryWeights$ in the ECML data set is illustrated in Figure 6 (upper). This histogram, representative of all the attack types, shows a great deal of numerical diversity (e.g., a few thousand unique data values for one vector in the dictionary). However, is this diversity truly necessary for accurate classification? Our hypothesis is that it is not, given that our application may only need a gross estimate of a term's relevance (i.e., "high, medium, or low").

Based on this hypothesis, we constructed a program that resamples or *quantizes* a dictionary's $tfidf$ scores across all attack types to a smaller number of unique data values. This approach employs a simple weighted

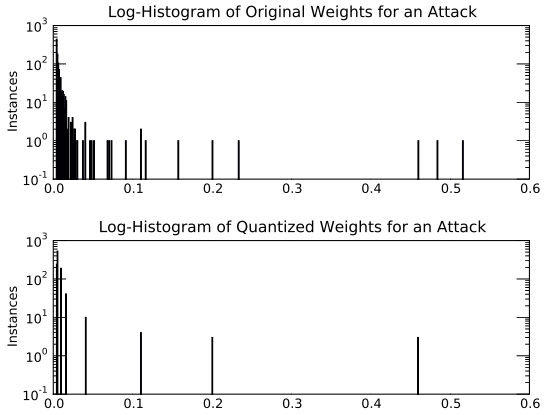

Figure 6: Log histogram of $tfidf$ score values

clustering algorithm that is weighted towards preserving larger data values. As illustrated in Figure 6 (lower), the number of unique data values is reduced to eight while maintaining a fair representation of the spectrum. Each of the nine vectors (idf and eight $categoryWeights$) in the dictionary are quantized individually. These data values are also transformed from a floating point representation to fixed point in order to simplify the hardware implementation.

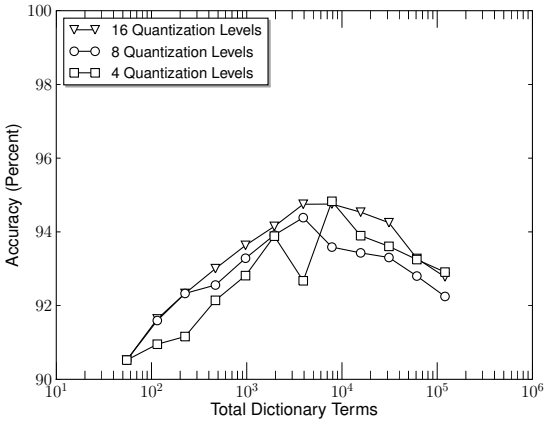

Figure 7: Impact of quantization vs. term size on accuracy

In order to test our hypothesis we generated a wide range of truncated,

quantized dictionaries from the training data set and evaluated accuracy when the classifier was applied to the testing data set. As illustrated Figure 7, reasonable accuracy can be achieved even when the dictionary is heavily quantized to contain just a few unique data values per vector. Over quantizing does result in instability and losses in accuracy.

5.2. Hash Methods

Terms in the original ECML training data set were on average 24 bytes long and in total were 19MB. Even with a reduced dictionary with only a few thousand terms, it is infeasible to store the original text in the dictionary. In addition to capacity issues, it is challenging to look up an entry in the dictionary of this size with minimal memory accesses, necessitating hashing. A plain hash table for dictionaries of this size will still not likely fit entirely in an FPGA's internal Block RAM. It is necessary to consider more probabilistic hash functions that estimate whether an input belongs to a set. Bloom filters [17] are a common technique for compactly implementing a set membership test. A Bloom filter consists of several hash functions and a bit vector. All hash functions are applied to an input term and the resulting hash values index into the bit vector at multiple locations. The term is considered a member of the set if all selected bits are set. The Bloom filter is a probabilistic technique as collisions may result in false positives, although there will be no false negatives. The number of hash functions and size of the bit vector may be configured to optimize between memory constraints and desired accuracy. Reducing the false positive rate requires more memory for the filter.

Our approach to implementing a quantized dictionary is to employ a large array of Bloom filters, with each filter representing a particular data value in the dictionary. At runtime an incoming term is hashed according to the needs of the Bloom filters. The hashes are dispatched globally and each Bloom filter tests whether the input is a member of its set. If a Bloom filter identifies a hit, the data value associated with the filter is presented to the corresponding scoring unit. While this approach does not scale when there are a large number of quantization levels or attack types, it does provide a compact means of housing a dictionary with a large number of terms.

In our initial implementation, we focused on combining C_1, C_2, and C_3 to minimize the amount of data required by the dictionary. While this approach worked, it suffered in accuracy because of both false positive rates and the lack of numerical diversity. Instead, implementing two statistics, *idf* and $categoryWeight$, in the dictionary provides a larger numerical range (i.e.,

multiply two 8-value numbers) and can cause better Bloom filter accuracy (i.e., a false positive must occur in both the *idf* and $categoryWeight$ lookup to propagate).

5.3. Hardware Layout

The layout of the top-level hardware design is illustrated in Figure 8. This architecture has five components.

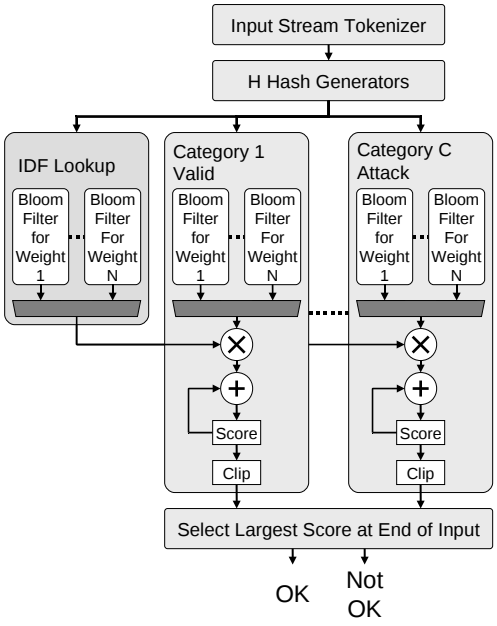

Figure 8: Top level hardware design

Input Stream Tokenizer The first unit in the data flow parses an input message from a queue and extracts a byte stream of lowercase tokens. This unit is the most complicated part of the design as tokens vary in length and are delimited by several character sequences. Tokenization is a serial operation that operates on byte-sized data values and is therefore the bottleneck in the design.

Hash Generators The second unit examines the incoming token byte stream and generates H different hashes for each token. A variety of hash functions were considered for this work. We ultimately selected a Pearson [18] hashing approach that employs $4 \times H$ randomly-generated 256-entry lookup tables to hash each token. To avoid hash collisions between small tokens, we inserted

a unit to append a token's bytestream with a 2-byte length. This unit adds two stall cycles per token to the byte stream, but greatly improves the quality of the hash functions.

IDF Lookup A single set of Bloom filters is used to perform a dictionary lookup of the input term's *idf* value. If the term is not found in any of the Bloom filters, an output of zero is produced. The design only requires a single IDF Lookup Unit, as the *idf* value for an input token is the same for all categories.

Category Analysis Units The bulk of the work in the design is performed by an array of category analysis units. Similarly to the IDF Lookup unit, a category analysis unit employs an array of Bloom filters to look up an input token's $categoryWeight$ for a particular attack type. This value is then multiplied by the *idf* value to compute the term's relevance, which is added to a cumulative score for the input message. When all tokens are processed, a threshold operation is applied to remove scores that do not meet a specified value. This threshold operation allows users to tune how sensitive the classifier is to malicious behavior.

Majority Vote The last unit in the dataflow examines the final scores of the different categories when all tokens are processed and selects the category with the largest value as the winner. The message is labeled as "ok" or "not ok" based on whether the winning classifier is the "valid" category or an attack category.

5.4. Generating the Hardware Classifier

An important aspect of this work is being able to rapidly generate custom hardware designs based on different input training data sets and user-selected parameters. This feature is essential in network security applications where new attack vectors and categories are added on a regular basis. Our approach to making a customizable hardware implementation is based on two components. First, a general-purpose hardware design was developed that is parameterized and can be adapted to different classification work based on updates to the Bloom filter data. Second, we developed a tool chain for automatically building hardware. As illustrated in Figure 9, the tool flow is based on several components.

- Training: A user can supply labeled training data to a TFIDF program to generate the full dictionary of TFIDF weights for the classifier. This data is exported to a SQLite database for data queries.

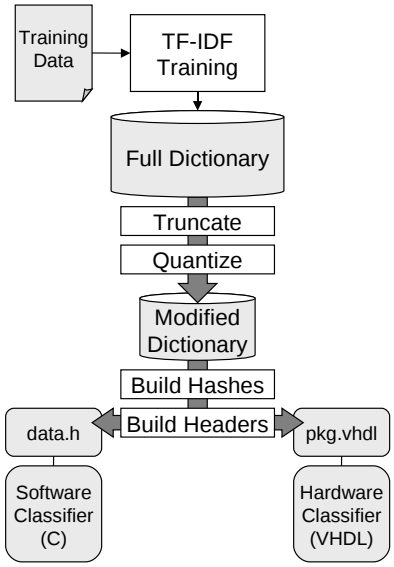

Figure 9: Tool Flow for Building Hardware

- Truncate: For each attack category, the top T terms and their statistics are extracted. The user selects the parameter T.

- Quantize: Each vector in the dictionary is run through a quantizer to reduce the number of unique data values in the dictionary. The number of quantization levels is a user-selected parameter chosen to trade between accuracy and memory footprint.

- Build Hashes: Data from the modified dictionary is then converted into a series of Bloom filters. The user may tune a Bloom filter error rate parameter to scale the memory footprint of the filters and the number of hash functions that are utilized.

- Build software: The tool chain can export hash data to a C header file. This file is utilized by validation tools (e.g., verify all dictionary tokens hash properly) and a stand-alone evaluation program (e.g., classify all inputs in a file).

- Build hardware: Finally, the tool chain constructs a VHDL package file that includes all the data necessary to instantiate the Bloom filters.

While the current approach requires the hardware design be recompiled when a new model is applied, it would be straightforward to allow updates to be completed through writes to the Bloom Filter Block RAMs.

5.5. Implementation Experiments

A number of experiments were conducted to validate both the hardware design and the tool chain. In all of the experiments we targeted a Xilinx Virtex 5-LX 50 part (XS5VLX50T-FG1136C-1) found on the Xilinx ML555 reference board. This part features sixty 36Kbit Block RAMs, allowing 240KB of 32b data values to be stored internally. We employed the ISE 11.1 tools and the built-in synthesis tool XST. For verification, a special design was constructed that supplied a number of input documents to the classifier. ChipScope was utilized to verify the output results were correct.

5.5.1. Utilization Characteristics

In order to observe how different parameters affect the hardware implementation, we constructed a reference design that simply instantiates an input FIFO, the classification core, and routes all of the I/Os to the FPGA's pins. This design does not serve as a functional system, but provides a means by which realistic implementations can be observed. We supplied a large number of configurations and measured the amount of resources required by each implementation.

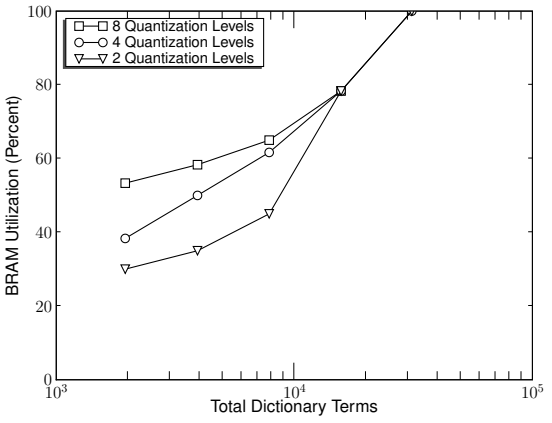

Figure 10: Memory footprint for different build parameters

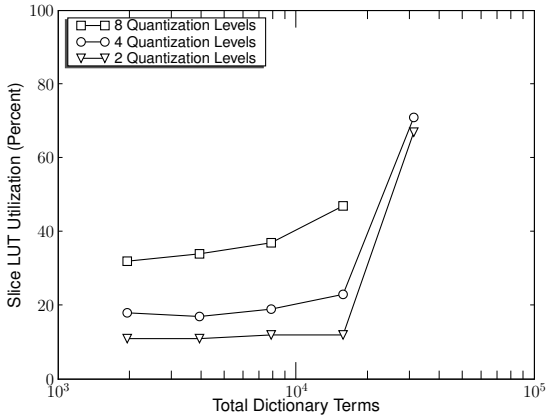

Figure 11: Slice utilization for different build parameters

Resource utilization numbers are presented in Figures 10 and 11. As expected, more quantization levels translates to more Block RAM utilization in the smaller designs. However, these memory requirements become less distinct as more terms are included in the dictionary. This trait can be attributed to the fact that Block RAMs are allocated in large capacities (2KB), and that the lower-term dictionaries do not fully utilize their Block RAM allocations (e.g., a 128B Bloom filter is implemented with a 2KB BRAM). In terms of slice utilization, the different designs remain relatively constant until Block RAM resources are fully consumed. While the ISE tools are sophisticated enough to switch to using slices as memory when Block RAM is exhausted, doing so rapidly fills the FPGA.

5.5.2. Performance Measurements

Performance of the hardware implementation depends on two factors: the average length of tokens in the input stream and the maximum rate at which the hardware can be clocked. For the former, each input contains a variable number of variable-length tokens. Our design processes data in a byte-stream manner and incurs two pipeline stalls at the end of each token encountered. For an input with C characters and T tokens, this delay results in a streaming efficiency of $\frac{C}{C+2 \cdot T}$. The design therefore has streaming efficiencies ranging from 0.5 in the worst case (when the input is a series of one character tokens) to nearly one in the best case (when the input is a single token). Inputs in both the ECML testing data set and the West Point data set were found to

provide an average streaming efficiency of 0.85.

For clocking measurements, we generated a design that employed eight quantization levels and a dictionary with 3,919 terms of statistics. We found that the maximum clock rate for this design was 196MHz. Multiplying the streaming efficiency of the ECML testing data set by this clock rate results in a streaming rate of 166MB/s. This data rate is sufficient for Gigabit network speeds and greatly outpaces software implementations without impacting accuracy.

6. Discussion

We have developed two very different streaming, parallel implementations of the HTTP attack classification algorithm on massively multi-core and FPGA processors. Our experienceis discussed along several dimensions.

Algorithm Development In terms of development effort, the Tilera algorithm took approximately three months for an engineer experienced in algorithm development for massively multi-core processors, but not with Tilera. The Linux/C programming environment made it possible to start with the base sequential program and gradually improve parallelism and performance. We found that the proprietary intrinsics for concurrency and communication were necessary to obtain high performance, but did not have to write assembly code. The relatively mature development tools and fast compile/debug process made it possible to code, test and analyze many different mapping and communications strategies.

The FPGA effort took six months, of which less than a month was spent in writing hardware description language code. The FPGA implementation was also undertaken by an experienced engineer with extensive hardware, software, and algorithm design and implementation experience. The most time consuming part of the FPGA version was in devising methods to reduce the model size without compromising accuracy, and in writing tools to help automate data analysis to arrive at appropriate quantization levels to encode the $CategoryWeights$. Standard, mature FPGA development tools from Xilinx were used to compile, simulate, and test the design.

Platform characteristics In addition to development time, the Tilera and FPGA co-processors can be compared according to cost and energy efficiency. In the form of PCIe co-processor boards, the two systems are roughly comparable in cost at roughly $2\times$ the cost of a standard workstation. They would be significantly less expensive than a workstation in quantity and

in embedded form factor. In terms of power, both platforms are low power compared to a workstation. The Tilera draws around 20W. The FPGA power draw depends on the design. Our design on the Virtex 5-LX uses about 1W.

Data set The characteristics of the dictionary and the streaming algorithms' throughput depend on the data set. Most of the analysis and experimental verification used the fully labeled ECML/PKDD data set, from which we analytically derived quantitative accuracy metrics. While we were not able to find another labeled data set, we also worked with data from the 2009 Inter-Service Academy Cyber Defense Exercise [6], also called the West Point data set. This data set consists of packet traces from network attack exercises by the National Security Agency on a network hosted by West Point Academy. Our algorithms detected HTTP attacks in this data set[2]. We ran a security module from the open source Apache web server over the West Point data set, and verified that this tool also flagged as attack the request packets flagged by our algorithm. Since the data set is not labeled, we could not generate a dictionary from the West Point data set, nor could we compute accuracy. As discussed in 5.5.2, the input term size can affect FPGA throughput. The input terms of the West Point data set had the same streaming efficiency as the ECML/PKDD, and therefore throughput was the same for both data sets.

Performance The streaming implementations were designed to optimize throughput with as little impact as possible on accuracy. Speed and accuracy for various implementations are summarized in Table 2.

The Java implementation of [2] focused on accuracy with a complete TFIDF dictionary and did not consider performance. On the Tilera, we reduced the size of the dictionary by retaining the top 255 high value terms so that an attack dictionary could fit completely in the L2 cache of each Tile64 core., which reduced accuracy by 2%. The combination of approximate state machine and reduced dictionary gave a throughput of 37MB/s. On the FPGA, we transformed the representation of scores in the dictionary from floating point to small fixed point values and used Bloom Filters to store those values, reducing the dictionary size to .2% of the original. Throughput on the ECML/PKDD and West Point data sets was 166 times the original.

To consider the effects of the Tilera and FPGA optimizations on con-

[2]We tried several other data sets, but they did not contain HTTP request attacks of this form.

Table 2: Throughput Comparison

Classifier	Throughput	Accuracy
Original Java	2MB/s	94%
Streaming Tilera (ST)	37MB/s	92%
Streaming HW (SHW)	166MB/s	94%
ST – workstation	10.2MB/s	92%
SHW – workstation	.901MB/s	94%

ventional CPUs, we also developed sequential C language implementations of the Tilera and FPGA algorithms. These versions were run on the Tilera host, a 2.2GHz x86 processor with 4GB memory. The results show that the Tilera optimizations also benefit standard workstation platforms. Although our workstation implementation is single threaded, we expect the algorithm to scale nearly linearly with additional cores. In contrast, the sequential performance of the FPGA algorithm is very poor. In this algorithm, each token is run through 32 byte-oriented hash functions. Each token requires 4 lookups into about 9×8 Bloom filters. These operations are friendly to logic gates, but inefficient for a 64-bit CPU.

7. Conclusions

In this work, we have described a novel algorithm to detect malicious web HTTP requests. We have shown two implementations of streaming classifiers capable of processing a text stream at 37MB/s and 166MB/s respectively. The classifiers detect seven different attack types and differentiate between attack and normal HTTP web page requests with an accuracy of 92% (Tilera) and 94% ((FPGA). Optimizations were employed to enable streaming, reduce computation, and minimize memory usage. The Tilera algorithm, implemented entirely in C and utilizing 55 processing cores, demonstrated throughput of 18.5X sequential software on a workstation. Even with a dictionary compressed to 0.2% the original, the FPGA hardware algorithm shows the same accuracy as the original software implementation, with 80X the throughput of the sequential algorithm. The performance and accuracy of our streaming classifiers allows them to be used as real time analysis components of an advanced intrusion prevention pipeline in network security

applications.

Acknowledgments The ECML/PKDD data set was obtained from the ECML/PKDD 2007 Workshop and is administered by Dr. Mathieu Roche. John May of LLNL designed and implemented an initial software streaming architecture and studied the impact of term frequency vector length on classification accuracy.

[1] G. Salton, A. Wong, C. S. Yang, A vector space model for automatic indexing, Communications of the ACM (11) (1975) 613–620.

[2] B. Gallagher, T. Eliassi-Rad, Classification of http attacks: A study on the ecml/pkdd 2007 discovery challenge (TR-414570).

[3] G. Salton, C. Buckley, Term-weighting approaches in automatic text retrieval, Information Processing and Management (1988) 513–523.

[4] C. Manning, P. Raghavan, H. Schutze, Introduction to Information Retrieval, Cambridge University Press, 2008.

[5] Ludovic Denoyer and Hung Son Nguyen, ECML/PKDD 2007 Discovery Challenge, Available http://www.ecmlpkdd.org/, 2007.

[6] ITOC, West point data set, http://www.itoc.usma.edu/research/dataset (2009).

[7] C. Rassi, J. Brissaud, G. Dray, P. Poncelet, M. Roche, M. Teisseire, Web analyzing traffic challenge: Description and results, in: Proceedings of the ECML/PKDD 2007 Discovery Challenge, 2007, pp. 47–52.

[8] K. Pachopoulos, D. Valsamou, D. Mavroeidis, M. Vazirgiannis, Feature extraction from web traffic data for the application of data mining algorithms in attack identification, in: Proceedings of the ECML/PKDD 2007 Discovery Challenge, 2007, pp. 65–70.

[9] I. Witten, E. Frank, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2005.

[10] M. Exbrayat, Analyzing web traffic: A boundaries signature approach, in: Proceedings of the ECML/PKDD 2007 Discovery Challenge, 2007, pp. 53–64.

[11] T. Chen, Z. Zheng, N. Zhang, J. Chen, Heterogeneous multi-core design for information retrieval efficiency on the vector space model, Fuzzy Systems and Knowledge Discovery, Fourth International Conference on 5 (2008) 353–357. doi:http://doi.ieeecomputersociety.org/10.1109/FSKD.2008.229.

[12] H. Song, J. W. Lockwood, Efficient packet classification for network intrusion detection using fpga, in: FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, ACM, New York, NY, USA, 2005, pp. 238–245. doi:http://doi.acm.org/10.1145/1046192.1046223.

[13] Z. K. Baker, V. K. Prasanna, Time and area efficient pattern matching on fpgas, in: FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, ACM, New York, NY, USA, 2004, pp. 223–232. doi:http://doi.acm.org/10.1145/968280.968312.

[14] A. Das, S. Misra, S. Joshi, J. Zambreno, G. Memik, A. Choudhary, An efficient fpga implementation of principle component analysis based network intrusion detection system, Design, Automation and Test in Europe Conference and Exhibition 0 (2008) 1160–1165. doi:http://doi.ieeecomputersociety.org/10.1109/DATE.2008.4484835.

[15] D. L. Olson, D. Delen, Advanced Data Mining Techniques, Springer, 2008.

[16] LLNL, Data-centric computing architectures, https://computation.llnl.gov/casc/dcca-pub/dcca/Downloads.html (2009).

[17] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.

[18] P. K. Pearson, Fast hashing of variable-length text strings, Commun. ACM 33 (6) (1990) 677–680. doi:http://doi.acm.org/10.1145/78973.78978.